SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 253–268 (1997)

Interview

Interview with Takashi Sano

TAKASHI SANO,[1] WEI-TEK TSAI[2]* AND SANJAI RAYADURGAM[2]

[1]*Software Factory Department, SE Technical Service and Support Division, Fujitsu Limited, 9-3 Nakase 1-chome, Mihama-ku, Chiba-city, Chiba 261, Japan*
[2]*Software Engineering Laboratory, Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, U.S.A.*

SUMMARY

In this interview, Takashi Sano reviews the origin and history of the year 2000 problem in Japan, and summarizes its status in Japan as of late 1996. He presents a way for software vendors to make this problem known to the software community because the problem is very serious, with far reaching impacts on an organization's economics, demand for talent and workforce scheduling. Then, he discusses the core techniques used at Fujitsu to tackle the year 2000 problem when a re-engineering approach is used. Finally, he provides an overview of a relevant software tool developed by Fujitsu. © 1997 by John Wiley & Sons, Ltd.

J. Softw. Maint., **9**, 253–268 (1997)

No. of Figures: 12. No. of Tables: 1. No. of References: 13.

KEY WORDS: year 2000 readiness; Japanese maintenance; Japanese software engineering; re-engineering; software tools; software maintenance

1. BIOGRAPHY

Takashi Sano is Director of the Software Factory Department of the Software Engineering Technical Service and Support Division of Fujitsu Limited in Japan, where he supervises programmers and analysts working on a variety of software projects. He is considered to be one of the most experienced software managers at Fujitsu, and has solved numerous practical problems in software development and maintenance. He has either lead or participated in more than one thousand large software projects in his career, whereas few software professionals in Japan have the opportunity to participate in more than a hundred large software projects in a lifetime. Furthermore, at the Fujitsu Software Factory, a project with one million lines of code ('LOC') is not considered a big project. Often a large project can be 15 million lines of code and super-large projects can be 70 million lines of code. He is currently leading Fujitsu's year 2000 project.

* Correspondence to: W.-T. Tsai, Department of Computer Science, University of Minnesota, 4-192 EE/CS Building, 200 Union St, S.E., Minneapolis, MN 55455, U.S.A. E-mail: tsai@cs.umn.edu

CCC 1040–550X/97/040253–16$17.50

Received 20 March 1997
Revised 8 May 1997

Mr. Sano visited the Software Engineering Laboratory at the University of Minnesota in October 1996. He spoke at length about the major re-engineering issues facing software maintainers, the year 2000 ('Y2K') problem, and the technologies used to address the Y2K problem. During his visit, Mr. Sano also demonstrated some software tools applicable to the Y2K problem.

This interview was conducted mostly in person during Mr. Sano's visit to the University of Minnesota. The interview was conducted by Dr. Wei-Tek Tsai. He edited the tape-recorded transcript translated by Sanjai Rayadurgam and himself, and supplemented the transcript with exchanges of e-mail with Mr. Sano. He and Sanjai Rayadurgam provided the Appendix. The edited interview transcript and Appendix were additionally edited by Ned Chapin to meet the *Journal's* requirements.

Mr. Takashi Sano can be reached by mail at the Software Factory Department, SE Technical Service and Support Division, Fujitsu Limited, 9-3 Nakase 1-chome, Mihama-ku, Chiba-city, Chiba 261, Japan. His e-mail address is: takashi@sf.tokyo.se.fujitsu.co.jp

The *Journal* has previously published papers providing some background on the software industry and software maintenance in Japan, such as the one by Pei and Tan (1994). The background data in the book edited by Matsumoto and Ohno (1989, pp. 257–277, pp. 303–320) may also be useful.

2. INTERVIEW

Tsai: Mr. Sano, please introduce yourself.

Sano: I was born and brought up in Hokkaido, a northern island of Japan. It is a very cold place, like Minnesota. I earned my B.S. from Hokkaido University. I majored in Precision Engineering at Hokkaido University. I was involved in the design of complex systems. In those days I worked with both digital and analogue computers. We used computers to solve differential equations. I was involved in analysing and studying aeroplane control systems. I continued my study for my M.S. in Control Engineering after finishing the B.S. program. After earning my M.S., I joined Fujitsu.

Until I joined Fujitsu, I never thought about software engineering or processes. When I entered Fujitsu, there was a language called COBOL that was new to me. I used to program in FORTRAN using `READ` and `WRITE` statements and mathematical equations, but file handling was new to me. In COBOL the concept of file is very important. In fact, it is a central concept. It took me about one month to learn COBOL. Nowadays, we use COBOL for client-server applications on Unix and Windows NT systems. Little did I know at that time that I would be working with COBOL programs for the rest of my life! These days, it is the most predominantly used language for banking, insurance and security applications in Japan. Many large corporations run their applications with more than a million of lines of code. I have been involved in over one thousand COBOL projects at Fujitsu.

Tsai: You are indeed a very experienced COBOL project manager. Before we talk about the Y2K problem could you tell us about how you obtain requirements from the customer?

Sano: Customers provide their requirements mostly in natural language. A semi-formal notation is used to specify the customers requirements by the analysts and designers. This is in the form of a table shown in Figure 1. We call this notation BR-SPEC (Business Rule SPECification). We specify the data items, the processing associated with the data items and the relevant conditions in a tabular format. We build in this notation in our integrated CASE tools, AA/BR-MODELER. At times the customers too provide the specifications in this form. A semi-automatic checking process is used to verify and validate the specifications. The syntax checking is automatic. These validated requirements are used to drive the later stages of design, coding and testing.

Tsai: What kind of CASE tools do you use?

Sano: The CASE tools that we use may be divided into three categories: upper-CASE, mid-CASE and lower-CASE. Upper-CASE tools use entity-relation notation and/or dataflow diagrams to model the business processes. Mid-CASE tools are used to specify the business rules. Lower-CASE tools generate the COBOL code for the application. Designers seldom read generated COBOL code, but only work with notations such as BR-SPEC.

Tsai: Could you now describe the Y2K problem as you see it?

Sano: The Y2K problem is a re-engineering issue. Many large business applications handle years as two digit numbers when processing dates. Only the last two digits of the year are stored and processed by these applications. By and large this did not cause any problems since the assumption that the first two digits are 1 and 9 was almost always correct. This assumption would fail once we have year dates in the 2000s. Systems that have been designed to handle dates this way would suddenly fail or produce erroneous results.

The seeds of this problem were sown a few decades back. When I started programming I used mini-computers. Later, when IBM OS/360 was released about 25 years ago, it became the industry standard. In those days, memory and hard drive capacity had a high premium. Engineers had to design their programs carefully to run them successfully with limited memory. 100MB of hard disk space was considered huge. But these days, a PC can have 100MB of main memory and several GB of hard disk space. The severe memory and hard disk constraints forced programmers to optimize their programs. For example, when we designed database layouts we had to be careful in allocating record sizes. Even a single byte may affect performance. This is the origin of the Y2K problem. One could hardly afford the extra storage required for a four-digit year. The COBOL standard at that time provided for only two-digit years. ANSI and ISO standards introduced a four-digit year notation only in 1989.

Data	Process	Condition

Figure 1. Format of the BR-SPEC form used to capture business rules

Many customs and practices also contribute to the Y2K problem. For example, when we write our birth year, we often use the last two digits rather than four digits. When I came to the U.S. recently to give a talk on the Y2K problem, I had to fill in an I-94 tourist form. That form had space only for two digits for the year in the birth date. Systems designed to handle such forms would treat years as two-digit data.

What complicates matters is the nature of the problem itself. While it is easy to explain the problem in a few simple words, it has the potential to bring the functioning of whole organizations to a grinding halt. In Japan, the fiscal year ends in March. Thus, by the end of March 99, when the 1999–2000 year starts, dates from 2000 would be used. Our customers have to modify their current two-digit year applications. Hence, they and we don't even have three complete years to take care of the Y2K problem.

Tsai: What is your responsibility concerning the Y2K problem?

Sano: At Fujitsu, the second largest computer company in the world, I am in charge of developing processes and tools to solve the Y2K problem. We have developed techniques and tools to re-engineer large COBOL programs affected by this problem. Starting from Spring 1996, we have been running a publicity campaign to inform and educate our customers about the importance of this problem.

Tsai: Do you have any data related to the Y2K problem? Specifically, do companies know about this problem and if they do, what are they doing to solve the problem?

Sano: I have some data on the problem collected from various surveys concerning Japanese corporations. I am afraid that I cannot speak about U. S. companies. Figure 2 shows the details of one such survey jointly conducted by Japanese Information Service Industry Association and Japan Users Association of Information Systems (JIAS, 1996).

Many companies have yet to start working on this problem. I understand that in the case of U. S. companies, some of them have worked on this problem, and some Japanese companies too have started addressing it. Unfortunately, many of them still have not paid it the attention it deserves.

Figure 3 shows that of the 750 users surveyed, only 7·7 per cent had completed their Y2K project. While 17·2 per cent have started work on the project, a large

JIAS/JUAS Survey 1996.8.10			
Objective	2,932 users	- Govt.	480 users
		- Priv.	2,452 users
Time	1996, May 10th ~ June 5th		
Result	750 users	- Govt.	174 users
	(25.6%)	Priv.	567 users

Figure 2. Details of the JIAS/JUAS survey

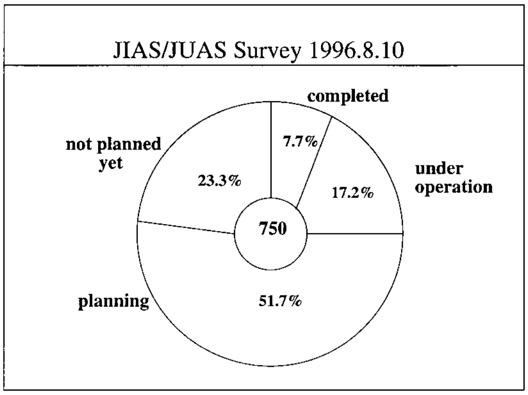

Figure 3. Y2K project status for non-government computer users

proportion of the users are still at the planning stage. What is more important is that 23·3 per cent have yet to start planning. All these projects must be executed in the next three years. This will require a lot of resources and will translate into a huge business for software maintainers. The survey also showed that the government users were lagging behind private organizations. The status chart for government users is shown in Figure 4.

Another important fact to be noted is that organizations are not viewing it as a single maintenance task. As shown in Figure 5, of the 58 completed projects only 6·9 per cent were outright Y2K projects. 29·3 per cent respondents had this as part of a bigger maintenance project and handled this with other maintenance issues. A sizeable 37·9 per cent executed this project as a reconstruction (re-engineering) project.

Even though they have only three years left, of the 692 users who have yet to complete the project, only 16·5 pr cent had completed the planning stage as of May 1996. Their responses showed that about 19·7 per cent of them expected

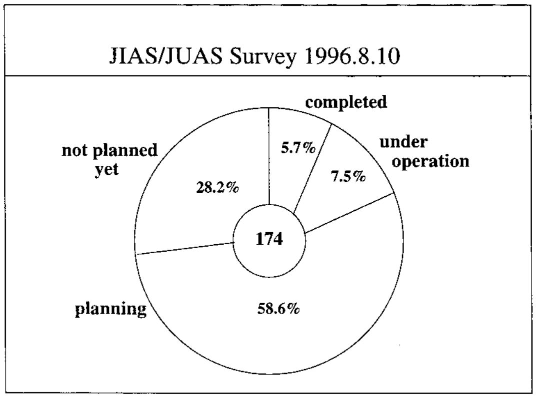

Figure 4. Y2K project status for government computer users

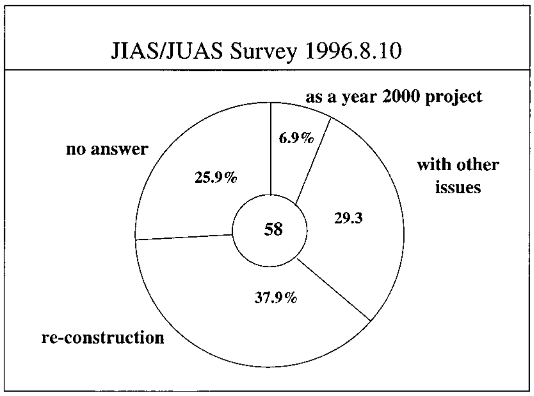

Figure 5. How the Y2K problem was approached in completed projects

to start planning during the second half of 1996, 26·6 per cent in 1997, and another 19·5 per cent in 1998. 18·2 per cent did not indicate when they would start planning. These results are shown in Figure 6.

The most critical data are those in Figure 7: survey respondents predict the peak activity period to be 1998. Around 40 per cent of the projects are expected to be in various stages of execution in 1998. These data are useful for managers to plan in advance how to schedule and allocate their resources. For a project like this, any delay would have a cascading effect on critical business functions of the client organizations. Further, for maintainers, this period is going to be critical. Since resources are expected to be stretched to the maximum during this peak period, there will be little tolerance for any snags in the project.

Tsai: How big is the Y2K business worldwide?

Sano: Very big indeed. I was invited to give a talk on the Y2K problem by the Information Technology Association of America (ITAA) for their 50th Anniversary Celebrations at Palm Springs in California. According to ITAA the world-

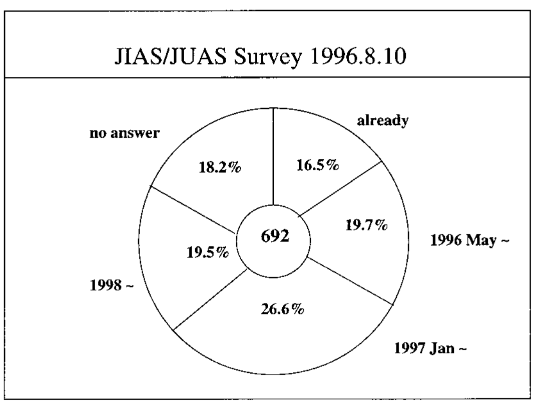

Figure 6. Planning status for Y2K projects as of May 1996

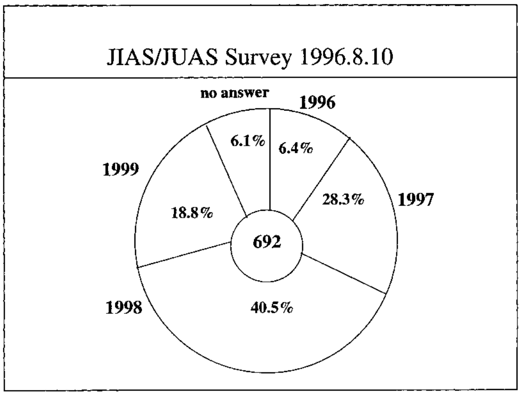

Figure 7. Peak period

wide business opportunity is about $600 billion. Previously, they had estimated that to be $300 billion, but recently they doubled their original estimate. From this estimate you can see that it is indeed a significant business.

Tsai: What are the typical applications in Japan that are affected by this Y2K problem?

Sano: A wide range of applications are affected by this problem. A simple program to sort dates with two digit years will not sort properly if the dates do not belong to 1900s. There are many mission-critical business applications that are affected. A car manufacturer has to plan in advance for production of parts for five or six years from the date of release of a new model. The production plan for parts of a car released in 1996 would run into 2000. Thus, the current production plan data would not be properly handled when two digits are used for years. Long-term services like banking and insurance have to schedule payments over a period of forty years and more. People working in these fields think about the problem in advance, but other industries like manufacturing and retail sales do not think about it in their design phase.

Tsai: How big are these applications?

Sano: We may classify applications of up to a million LOC as small. Large applications may be from 12 to 15 million LOC in size. There are also super-large applications of size 70 MLOC and more. Many large and super-large applications are affected by this problem.

Tsai: Are all these applications written in COBOL? What about languages like C++?

Sano: Languages like C++ and its associated tools like Visual C++ are seldom used in mission-critical systems. Their size may be around 100 to 200 thousand LOC. COBOL is a business-orientated language. Many large systems have been implemented in COBOL; many industrial people currently use COBOL; and now COBOL is applicable on servers such as Unix and Windows NT.

Tsai: What is the average cost of re-engineering a typical application?

Sano: According to one estimate the cost to change and test in re-engineering in the U.S. is between $1.00 and $1.50 per statement. The corresponding cost for Japan would be between $0.80 and $1.50 per LOC. A batch application may have 500

to 1000 LOC. An on-line banking system may have about 10 000 LOC. Re-engineering may involve 500 000 LOC for reasonably large applications. For super-large applications re-engineering is neither economical nor practical within the limited time available for handling Y2K problems.

Tsai: How serious is the problem of finding human resources for the handling the Y2K problem?

Sano: It is very serious. In Japan, as in the U.S., most universities train their graduates in C/C++ or other languages, in fact in anything but COBOL. Internet, Groupware, Client/Server, C++, RAD (Rapid Application Development), OO (Object-oriented), Visual Basic and Power-builder are currently considered hot skills. COBOL is not in this list. But the number of COBOL programmers is dwindling over the years. This makes me believe that the situation will change. Very soon COBOL will have a good market value. The Y2K problem is in a way hastening this change. Many large programs that need to be re-engineered are written in COBOL, DB2, CICS and IMS. About 90 to 95 per cent of existing customer software assets are on mainframes written in these languages and to some extent in PL/1, in Japan. Hence there will be great demand for these programmers.

Tsai: How do software companies address the Y2K problem?

Sano: Each company has its own strategy to address the Y2K problem. All major multinationals in Japan, such as IBM, Hitachi, NEC and Fujitsu are working on the problem. IBM has a web page for its Y2K technical support centre at: http://www.software.ibm.com/year2000/ and Fujitsu has a web page at: http://www.fujitsu.co.jp/hypertext/2000/

I can talk only about Fujitsu. We started off with a survey of our own customers to find the current status of their Y2K projects. The survey indicated that more than 50 per cent had done nothing about it. Also about 50 per cent of the customers did not have any idea about when to start the project. They considered the Y2K problem as a *future* subject. We concluded that we needed to make our customers understand the problem. Awareness about the problem would help both Fujitsu and our customers. By thinking and planning in advance, they would be more likely to make a smooth transition. Having customers plan ahead would help us in project scheduling and resource planning. We then would stand a better chance of successfully completing the project in time.

There is a basic difference between the U.S. model and the Japanese model in the information technology business. In the U.S., while computer companies sell the hardware and OS (operating system) software, application software and integration are handled by Independent Software Vendors (ISVs) as diagrammed in Figure 8. Customers directly procure the required applications from the ISVs.

In Japan the business structures are different, as diagrammed in Figure 9. We have what are called software houses. All major computer companies are situated in and around Tokyo. Fujitsu provides complete solutions to its customers, from the hardware and OS to customized application software. Fujitsu handles system integration and provides customized solutions to its clients. Thus, Fujitsu interacts with the software houses to find the right applications for its customers. It is

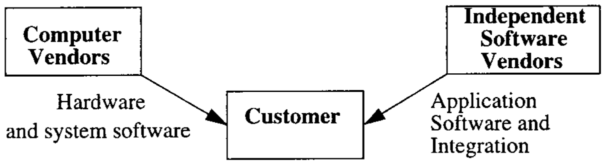

Figure 8. The U.S. business model for software acquisition

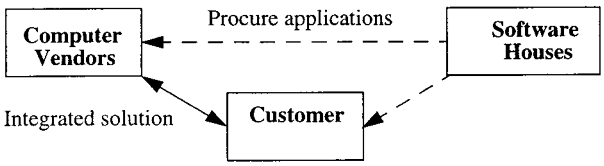

Figure 9. The Japanese business model for software acquisition

said that only about 11 per cent of Japanese users procure packaged software, the others procure customized software.

Tsai: What is the life cycle of a typical re-engineering process?

Sano: The re-engineering process follows an *analysis–design–modify–test* cycle. The time spent on a typical project can be roughly divided as follows:

- 30–40 per cent in analysis and design,
- 10–20 per cent in modification, and
- 40–50 per cent in testing.

These may vary depending on the project complexity and the customer's knowledge about their applications.

Tsai: A substantial part of the time is spent in testing. Do you think this can be improved?

Sano: The time spent in various stages depends on the customer's application knowledge. Some customers have 10–15 year old documentation. If the documentation with the customer is 100 per cent accurate then we may be able to improve the testing time spent. But I don't see this happening in practice.

Tsai: So, you don't see any significant reduction in testing time. What are the other difficulties faced in a re-engineering project like this?

Sano: There are many issues in Y2K re-engineering projects like this. I will discuss the major problems here, such as:

- no or inadequate inventory of software,
- different versions of languages used,
- weak documentation,
- varied coding conventions, and
- number and variety of external interfaces.

Lack of a proper inventory of programs is a major problem. In a typical project, about 20 to 30 per cent of the programs are not used at all. These

statistics can be found by analysing JCLs and source code. Of the existing code, 60 to 70 per cent is for error handling. We may be able to safely delete a substantial part of the existing modules, but many users think that they still need all the modules. A typical customer's system lives on and on for a lifetime. Newcomers never know whether they can discard some portions or modules safely without affecting the system.

Different versions of the COBOL languages are in use. In the U.S. there are applications written in COBOL 85, COBOL 74 and COBOL 68. Applications have different sub-systems written in different versions of the language. This at times may be an issue.

The correctness and completeness of documentation has a major impact. At times, documents are written in *pencil*. The writing fades away with time making it difficult to read. The remedy here is to generate documentation from the source code, but there are times when even the source code is not available (only the object code is available). The only possible course is to reconstruct and revalidate the documentation by testing the software.

Coding conventions affect program understanding. Customer systems that evolve over a period of time follow different coding conventions at different stages. It may be difficult to understand such programs. This would hamper maintenance and re-engineering of those programs at one glance.

External interfaces require careful examination. Modifications that propagate across external interfaces are difficult to handle.

Tsai: Can you briefly explain the Y2K re-engineering process that you follow at Fujitsu?

Sano: We follow a data-centred approach for handling the Y2K problem (see the Appendix for supporting background). The key idea is to first identify the Y2K-related data items by domain analysis and then use ripple effect analysis to identify other indirectly related variables in the program.

There are five major steps in the re-engineering process:

- inventory creation,
- dictionary creation,
- identification of Y2K data items by ripple effect analysis,
- modification, and
- verification.

As I mentioned earlier, lack of inventory is a major problem affecting re-engineering projects. The first step is to make inventory of all software artefacts like programs and documents associated with the application. This will help maintainers identify the components of the software that are relevant and remove obsolete parts.

The second step is the creation of a data dictionary. This stores detailed information about all the data items for each program. The description includes the name of the data item and its type, which are automatically extracted from the COBOL source code. It also has links to statements that use or define the data item. This information is obtained from a dependence graph obtained as a

result of dependence analysis of the source code.

The starting point for modifications is the identification of data items related to Y2K by using domain analysis. Naming conventions and data type information are used to identify the data items. For example the naming convention may show that variables having YY or YMD as a suffix store the year. Using this convention, SALE-YY would be identified as a Y2K-related variable.

The next step is ripple-effect analysis. Starting points for ripple-effect analysis are picked up from the data dictionary and/or from file layouts. Ripple-effect analysis is performed to identify all the Y2K-related data items. This finds other variables that either directly or indirectly depend on the list of initial variables. For example, if a variable TEMP is assigned a value from the variable SALE-YY, the output of ripple-effect analysis on the source program starting with SALE-YY would identify TEMP. The tool would identify both SALE-YY and TEMP as Y2K-related variables as shown in Table 1. Experiments on several large applications have shown that the Fujitsu tool is very effective in identifying Y2K data. Some application specific modifications are done. It must be noted that ripples may span multiple programs within the application.

After identification, the next step is to effect the modification required and desk validate. Data conversion and data migration are performed in this step. Apart from the Y2K data, programs that manipulate these data may also have to be changed. The affected program statements are obtained by starting with the list of affected variables and performing a dependence analysis. The modified code is then tested to validate the changes. This cycle repeats for each program in the inventory that has Y2K relevant data.

A *diff* tool is used to compare the old and new versions of the source code and its output is stored so that maintainers and testers can refer to it. The Year 2000 tool uses a very stable database system called the Object Store. This is an object-orientated database system. All the Y2K project-related data are stored in Object Store.

Figure 10 provides a high level view of the various support tools and the phases in which they are used.

Tsai: Y2K is only one of the many problems amenable to a re-engineering approach. Can these techniques and tools be used on other problems too?

Sano: Yes. There are many other similar re-engineering problems which can be effectively addressed using these tools. In Japan the number of digits in the postal codes will be increased soon. The telephone numbers too will change. Various laws have an impact on information systems. Vehicle registration and

Table 1. Identification of Y2K data

COBOL source code	Y2K variables	
MOVE SALE-YY TO TEMP	SALE-YY	by naming convention
	TEMP	by ripple-effect analysis

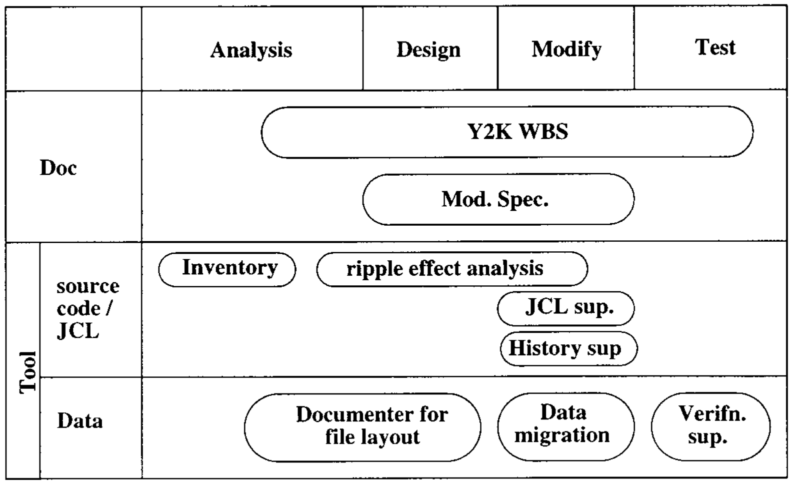

Figure 10. Suite of software tools used at Fijitsu for the Y2K problem

consumption tax are examples of these. Information systems for the 21st century have to be built today by reviewing the existing systems and renewing them. The Y2K problem is a trigger to initiate the renewal process. To be successful, the project must be started as soon as possible and executed within the restricted period according to schedule. We have a scant three more years to go!

Tsai: Thank you, Mr. Sano.

Sano: You are welcome.

APPENDIX: KEY TECHNOLOGIES USED BY FUJITSU

A1. Software engineering

Large business applications written in COBOL typically involve much more data movement and manipulation than sequencing, branching or looping. In other words, these programs tend to be relatively data-rich and control-poor. Analysing relationships between data is crucial for re-engineering such programs. Joiner *et al*. (1994) proposed a *data-centred* approach for analysing data relationships. This is discussed in the following sections. Fujitsu's Year 2000 tool uses this data-centred approach. Yoshino *et al*. (1995) discussed a reverse-engineering technology used by Fujitsu in generating narrative specifications from source code. Readers interested in a broader understanding of the Japanese software engineering practices are referred to Matsumoto and Ohno (1989) and Pei and Tan (1994).

A2. Data-centred approach

Maintainers trying to re-engineer large business applications may first wish to understand data and their interrelationships before understanding paths along computations from one

set of variables to another. The data-centred approach offers distinct advantages over the control-based approach with respect to data-related queries. It maintains dependence relationships among variables explicitly. In contrast, in a control-based approach such relationships are computed from path-relationships which is time-consuming. Huang *et al.* (1996a) and Chen *et al.* (1996) are examples of the usage of this data-centred approach.

The data-centred approach begins by automatically reverse engineering a set of dependence graphs directly from the source code. One such example is the Module-level COBOL Dependence Graphs (MCDGs) (Ferrante, Ottenstein and Warren, 1987). These capture the relationships that exist among various program entities such as variables, modules and files. The MCDG is a directed graph whose edges represent data and control dependence between variables, literals, modules and files which are represented as vertices. Joiner *et al.* (1994) describe a method of constructing MCDGs from COBOL source code. Figure 11 is an example of MCDG. The value of variable C depends on the values of A and B. EC is similarly dependent on the constant +99 but control reaching this point is conditional on SIZE ERROR and hence it is control dependent on A and B.

Algorithms of the data-centred approach rely on the dependence relationships as captured in the MCDGs. The main features of a data-centred approach (Joiner and Tsai, 1993) include variable classification, generalized slicing and ripple-effect analysis, which are briefly discussed in the next three subsections.

A3. Variable classification

This involves classifying program variables into different categories. It is the first step in understanding data-rich programs. These programs have hundreds or thousands of variables. It is difficult to find the variables of interest without a clear way of organizing them into different categories. Variables in COBOL programs may be classified into eight different categories (Chen *et al.*, 1994): domain, program, local, global, input, output, constant and control. Such a classification provides information about the role of a given variable in a program. Many techniques are available to classify variables. These range from simple variable-name based classification to automated techniques that use classification rules based on the language and domain. Chen *et al.* (1994) gave a completely automated classification technique. It is based on a set of classification rules and uses the dependence relationships as captured in the MCDGs to completely classify all variables

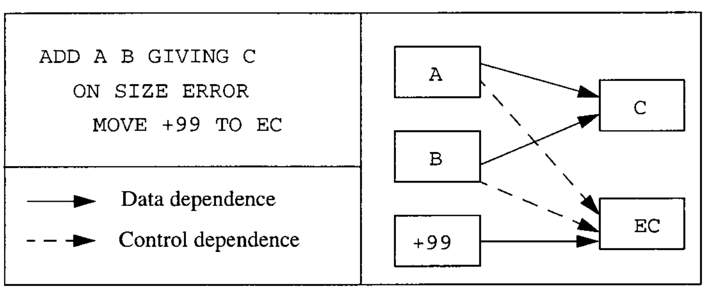

Figure 11. An MCDG for an example of COBOL source code

in a COBOL program. These techniques can be extended to identify Y2K-related variables as done by Fujitsu's Year 2000 tool.

A4. Generalized program slicing

Once the variables of interest are identified, the maintainer may wish to identify parts of the program that either affect or are affected by these variables. This is achieved by slicing.

Program slicing is a technique to decompose programs based on data and control flow information (Weiser, 1984). It starts with a slicing criterion specified as a tuple $C = \langle i, V \rangle$, where i is a program statement and V is a set of variables. The output of a slicing operation is called a slice which is a set of statements that might affect the value of the variables in V at the statement i. This was later extended to forward slicing where the output is the set of statements that might be affected by the value of the variables in the given statement.

The usefulness of program slicing and forward slicing is limited when they are used for large software systems, since the slices produced are often huge and hence difficult to manage. To overcome such limitations, Huang, Tsai and Subramanian (1996b) proposed a generalized program slicing technique. The slicing criterion is now specified as: $C = \langle i, V, Cons, d \rangle$. V and i have the same meaning. *Cons* is a set of constraints which can be used to prune or limit the set of statements selected and d is the direction of slicing. Generalized program slicing can be used to obtain focused slices from large programs. Slicing algorithms for COBOL programs can be performed by using the dependence information captured in the MCDGs. Dependence edges are travelled in a forward or backward direction to find the slice.

A5. Ripple-effect analysis

The maintainer would like to ensure that changes made to a program do not have undesirable side effects. Ripple-effect analysis is a process to ensure consistency and integrity after changes are made to the software. It is an iterative process. Modification to one part of a program may trigger changes to other parts to maintain consistency and remove side effects.

Figure 12 describes a generic four-step ripple-effect analysis process. Each iteration starts with an initial change. The second step identifies parts of the program (called potential ripples) that are potentially affected by the initial change. The third step is to determine which of these are actual ripples that require further change. The fourth step is to determine how to make these changes. The process repeats again with each new change until no more changes are required.

Ripple-effect analysis can be performed using generalized program slicing techniques. Wang, Tsai and Rayadurgam (1996) examined the role of slicing in ripple-effect analysis and established the need for generalized slicing techniques like constraint slicing and hierarchical slicing.

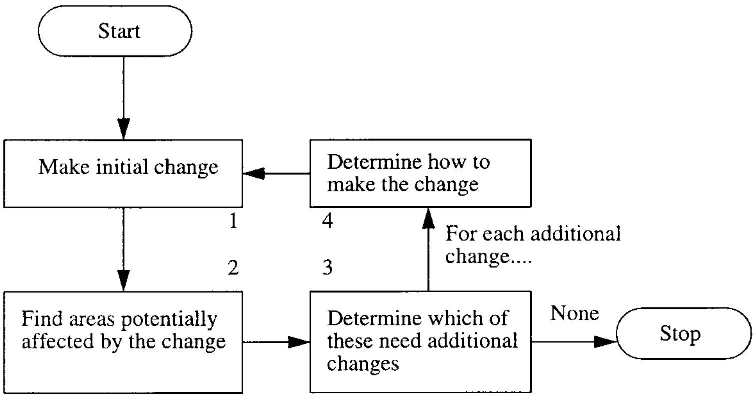

Figure 12. Summary of ripple-analysis procedure

References

Chen, X. P., Tsai, W.-T., Huang, H., Poonawala, M., Rayadurgam, S. and Wang, Y. (1996) 'Omega—an integrated environment for C++ program maintenance', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 114–123.

Chen, X. P., Tsai, W.-T., Joiner, J. K., Gandemaneni, H. and Sun, J. (1994) 'Automatic variable classification for COBOL programs', in *Proceedings COMPSAC94*, IEEE Computer Society Press, Los Alamitos, CA, pp. 432–437.

Ferrante, J., Ottenstein, K. J. and Warren, J. D. (1987) 'The program dependence graph and its use in optimization', *ACM Transactions on Programming Languages and Systems*, **9**(3), 319–349.

Huang, H., Tsai, W.-T., Bhattacharya, S., Chen, X. P., Wang, Y. and Sun, J. (1996a) 'Business rule extraction from legacy code', in *Proceedings COMPSAC96*, IEEE Computer Society Press, Los Alamitos, CA, pp. 162–167.

Huang, H., Tsai, W.-T. and Subramanian, S. (1996b) 'Generalize program slicing for software maintenance', in *Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering*, Knowledge Systems Institute, Skokie, IL, pp. 261–268.

JIAS (1996) *JIAS/JUAS Survey 1996.8.10*, Japanese Information Service Industry Association, Tokyo, 56 pp.

Joiner, J. K. and Tsai, W.-T. (1993) *Ripple Effect Analysis, Program Slicing and Dependence Analysis*, TR-93-94, Computer Science Department, University of Minnesota, Minneapolis, MN, 30 pp.

Joiner, J. K., Tsai, W.-T., Chen, X. P., Subramanian, S., Sun, J. and Gandemaneni, H. (1994) 'Data-centered program understanding', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 272–281.

Matsumoto, Y. and Ohno, Y. (Eds) (1989) *Japanese Perspectives in Software Engineering*, Addison-Wesley Publishing Company, Reading, MA, 326 pp.

Pei, G. and Tan, V. (1994) 'Reusability in software maintenance: a Japan–USA comparison', *Journal of Software Maintenance*, **6**(4), 165–183.

Wang, Y., Tsai, W.-T. and Rayadurgam, S. (1996) 'The role of program slicing in ripple effect analysis', in *Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering*, Knowledge Systems Institute, Skokie, IL, pp. 369–376.

Weiser, M. (1984) 'Program slicing', *Transactions on Software Engineering*, **SE-10**(4), 352–357.

Yoshino, T. S., Uehara, S., Ookubo, T., Suguta, S., Hotta, Y. and Sonobe, M. (1995) 'Reverse

engineering from COBOL to narrative specification', in *Proceedings COMPSAC95*, IEEE Computer Society Press, Los Alamitos, CA, pp. 284–289.

